Evaluation of an Eager Protocol Optimization for MPI

Ron Brightwell and Keith Underwood

Center for Computation, Computers, Information, and Mathematics
Sandia National Laboratories[⋆]
PO Box 5800
Albuquerque, NM 87185-1110
{rbbrigh,kdunder}@sandia.gov

Abstract Nearly all implementations of the Message Passing Interface (MPI) employ a two-level protocol for point-to-point messages. Short messages are sent eagerly to optimize for latency, and long messages are typically implemented using a rendezvous mechanism. In a rendezvous implementation, the sender must first send a request and receive an acknowledgment before the data can be transferred. While there are several possible reasons for using this strategy for long messages, most implementations are forced to use a rendezvous strategy due to operating system and/or network limitations. In this paper, we compare an implementation that uses a rendezvous protocol for long messages with an implementation that adds an eager optimization for long messages. We discuss implementation issues and provide a performance comparison for several micro-benchmarks. We also present a new micro-benchmark that may provide better insight into how these different protocols effect application performance. Results for this new benchmark indicate that, for larger messages, a significant number of receives must be pre-posted in order for an eager protocol optimization to outperform a rendezvous protocol.

1 Introduction

Nearly all implementations of the Message Passing Interface (MPI) employ a two-level protocol for implementing the peer communication functions. This strategy is an attempt to optimize short messages for latency and long messages for bandwidth. For short messages, data is eagerly sent along with the MPI envelope information (context, tag, etc.). This allows the send operation to complete without any interaction with the receiver. The long message protocol is typically implemented using a rendezvous mechanism where the sender must first send a request and receive an acknowledgment before the data can be transferred. Since the message to be sent is large, the overhead of the protocol exchange with the receiver is amortized by the transfer of the data.

While there are several possible reasons for using a rendezvous protocol for long messages, most implementations are forced to use it due to operating system and/or network limitations. In the case of a network that uses remote DMA to transfer a long message, the receiver may have to perform resource management activities, such as

[⋆] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

pinning down memory pages, and provide the sender with information necessary to start the data transfer. One machine where there are no such restrictions is the ASCI Red machine at Sandia National Laboratories. The production implementation of MPI for this machine uses eager sends for all message sizes. We believe that this approach has some benefits, and previous research has demonstrated a significant performance gain for using an eager protocol for larger messages for certain benchmarks [7]. In order to analyze the effect of using eager sends for all messages, we have modified the implementation to use a standard rendezvous protocol for large messages.

In this paper, we compare the strategy of using a rendezvous protocol with an eager optimization with a standard non-eager rendezvous protocol. We discuss implementation issues and provide a performance comparison for several micro-benchmarks. In addition, we present a new micro-benchmark that we have developed that may provide better insight into the effect of these different protocols on application performance. This new micro-benchmark varies the number of pre-posted receives to show the impact of unexpected messages on bandwidth.

The rest of this paper is organized as follows. The following section describes the hardware and software environment of ASCI Red, including the original MPI implementation. Section 3 describes the new rendezvous implementation. In Section 4, we discuss the micro-benchmarks that were run and present performance results and analysis. We discuss the limitation of these benchmarks and present a new micro-benchmark together with performance results in Section 5. Section 6 summarizes the important results, and we conclude in Section 7 with a discussion of future work.

2 ASCI Red

The Sandia/Intel ASCI/Red machine [6] is the United States Department of Energy's Accelerated Strategic Computing Initiative (ASCI) Option Red machine. It was installed at Sandia National Laboratories in 1997 and was the first computing system to demonstrate a sustained teraFLOPS level of performance. The following briefly describes the hardware and system software environment of the machine and discusses some unique features that are relevant to this particular study.

2.1 Hardware

ASCI/Red is composed of over nine thousand 333 MHz Pentium II Xeon processors. Each compute node has two processors and 256 MB of memory. Each node is connected to a wormhole-routed network capable of delivering 800 MB/s of bi-directional communication bandwidth. The network is arranged in a 38x32x2 mesh topology, providing 51.2 GB/s of bisection bandwidth. The network interface on each node resides on the memory bus, allowing for low-latency access between host memory and the network.

Despite its age, we feel that the ratio of peak network bandwidth to peak compute node floating-point performance makes ASCI Red a viable platform for this study. We believe this balance to be an important characteristic of a highly scalable machine, and few machines exist today that exhibit this level of balance. Clusters composed of commodity hardware, for example, have floating-point performance that greatly exceeds their network capability.

2.2 Software

The compute nodes of ASCI/Red run a variant of a lightweight kernel, called Puma [4], that was designed and developed by Sandia and the University of New Mexico. A key component of the design of Puma is a high-performance data movement layer called Portals. Portals are data structures in an application's address space that determine how the kernel should respond to message passing events. Portals allow the kernel to deliver messages directly from the network to application memory. Once a Portal has been set up, the kernel has sufficient information to deliver a message directly to the application. Messages for which there is no corresponding Portal description are simply discarded. Portals also allows for only the header of a message to be saved. This keeps a record of the message, but the message data is discarded.

The network hardware, the lightweight kernel, and Portals combine to offer some very unique and interesting properties. For example, messages between compute nodes are not packetized. An outgoing message to another compute node is simply streamed from memory onto the network. The receiving node determines the destination in memory of the message and streams it in off of the network. Also, reliability is handled at the flit level at each hop in the network, but there is no end-to-end reliability protocol. The observed bit error rate of the network is at least 10^{-20} eliminating the need for higher-level reliability protocols.

2.3 MPI Implementation

The MPI library for Portals on ASCI Red is a port of the MPICH [2] implementation version 1.0.12. This implementation was validated as a product by Intel for ASCI/Red in 1997, and has been in production use with few changes since. Because this paper focuses on long messages, we only describe the long message protocol in this paper. See [1] for a more complete description of the entire implementation. In the rest of this paper, we will refer to this protocol as the eager-rendezvous protocol.

When the MPI library is initialized, it sets up a Portal for receiving MPI messages. Attached to this Portal is the posted receive queue, which is traversed when an incoming message arrives. At the end of this queue are two entries that accept messages for which there is no matching posted receive. The first entry handles short messages, while the second entry handles long messages. The entry for long messages is configured such that it matches any incoming long protocol message and deposits only the message header into a list. The corresponding message data is discarded. An acknowledgment is generated back to the sender that indicates that none of the data was received.

For long protocol sends, the sender first sets up a Portal that will allow the receiver to perform a remote read operation on the message. The sender then sends the message header and data eagerly. If a matching receive has been pre-posted, the message is deposited directly into the appropriately user-designated memory, and an acknowledgment indicating that the entire message was accepted is delivered to the sender. The sender receives the acknowledgment and recognizes that the entire message was accepted. At this point the send operation is complete.

If the message was unexpected, then the incoming message will fall into the long unexpected message entry described above. In this case, the sender receives the acknowledgment and recognizes that the entire message was discarded. The sender must then wait for the receiver to perform a remote read operation on the data. Once the receiver has read the data, the send operation completes.

There are several reasons why we chose to implement an eager-rendezvous protocol for long messages. Portals are based on the concept of expected messages, so we wanted to be able to take advantage of that feature as much as possible. We wanted to reward applications that pre-post receives, rather than penalizing them. We believed that we were optimizing the implementation for the common case, although we had no real data to indicate that long messages were usually pre-posted. Because the network performance of ASCI Red is so high, the penalty for not pre-posting receives is not that large. The compute node allocator is topology-aware, so jobs are placed on compute nodes in such a way as to reduce network contention as much as possible. The impact of extra contention on the network resulting from sending a large unexpected message twice (once when the message is first sent and once again when the receiver reads it) is minimal. As mentioned above, previous research results had indicated that an eager protocol could provide significant performance benefits. Finally, using an eager protocol insures that progress for non-blocking messages is made regardless of whether the application makes MPI calls. The implementation need not use a separate progress engine, such as a thread or timer interrupt or require the application to make MPI library calls frequently, to insure that asynchronous message requests make progress. This greatly simplifies the MPI implementation.

3 Standard-Rendezvous Implementation

In this section, we describe the standard-rendezvous implementation. We have added this capability to the existing eager-rendezvous implementation. The standard-rendezvous protocol can be selected at run time by setting an environment variable, so there is no need to re-link codes.

On the send side, the rendezvous implementation essentially skips the step of sending data when a long send is initiated. It also does not wait for an acknowledgment from the receiver. It sets up a Portal that will allow the receiver to perform a remote read operation, sends a zero-length message, and waits for an indication that the buffer has been read by the receiver. While this change may appear to be minor and straightforward, it has a large impact on how completion of message operations is handled.

Assuming an MPI application with two processes, we examine the two rendezvous protocols for the following legal sequence of calls on both processes:

```
MPI_Irecv( buf, count, datatype, destination, tag, communicator, request );
MPI_Send( buf1, count, datatype, destination, tag, communicator );
MPI_Wait( request, status );
```

For the eager-rendezvous protocol, the `MPI_Irecv()` call sets up the necessary match entry on the receive Portal (assuming the message has not arrived) and returns. The `MPI_Send()` call sends the message eagerly and waits for an acknowledgment. Because the receive was pre-posted, the entire message is deposited into the receive buffer, an acknowledgment indicating this is generated, and the send completes when this acknowledgment is received. The `MPI_Wait()` call waits for the incoming message to arrive. Once the entire message is received, the wait call completes. Alternatively, if the send is unexpected (on either process), the receive call will find the unexpected message and read it from the sender. The send call will wait for the receiver to pull the message before completing. The wait call will block until the entire message is pulled. In all of these scenatrios, the MPI implementation is only concerned with initiating or completing the specific request made by the function call. That is, completing a send or receive operation only involves that specific send or receive operation.

In contrast, the standard-rendezvous implementation is more complex. If the receive call is pre-posted, the send call only sends a message indicating that it has data to send. In order to complete the send call, the receiver must receive this request and pull the data from the sender. This means that a send call cannot simply block waiting for the receiver to pull the message. It may itself have to respond to an incoming send request. In the above code example, if both processes block in the send call waiting for a response from the receiver, both will deadlock. Instead, the implementation must maintain a queue of posted receives. The send call must continuously traverse this queue to see if any posted receives can be progressed. In general, the implementation cannot just try to complete the incoming request from the function call. It must be concerned with *all* outstanding send and receive requests. This can be less efficient, since Asynchronous messaging requests only make progress when the application calls any of the test or wait family of functions.

4 Micro-Benchmark Performance

This section presents the micro-benchmarks used and compares the performance of the two MPI implementations. All of the tests were run in the interactive partition of ASCI Red. This partition is set aside for short runs associated with development. There is some impact of running these benchmarks on a shared machine, but repeated runs showed this impact to be minimal.

The first micro-benchmark is a ping-pong bandwidth test. The sender and receiver both pre-post receives and then exchange a message. The bandwidth is based on the time to send the message and receive the reply. Bandwidth is calculated by averaging the results of repeated transfers. Figure 1(a) shows the ping-pong bandwidth performance. The eager-rendezvous protocol outperforms the standard-rendezvous protocol by nearly 50 MB/s for 5 KB messages and 15 MB/s for 100 KB messages.

The second micro-benchmark is the Post-Work-Wait (PWW) method of the COMB [3] benchmark suite. This benchmark calculates effective bandwidth for a given simulated work interval, which measures bandwidth relative to the host processor overhead. Since PWW is aimed at measuring the achievable overlap of computation and communication, it uses pre-posted receives to allow the MPI implementation to make progress on outstanding operations. Figure 1(b) shows the PWW performance of the two protocols for 50 KB, 100 KB, and 500 KB messages. Again, the eager-rendezvous outperforms the standard-rendezvous for all message sizes. For 500 KB messages, the difference is slight. PWW shows less of a difference than the ping-pong benchmark.

The third micro-benchmark, NetPIPE [5], measures aggregate throughput for different block sizes. It also uses a ping-pong strategy, but it determines bandwidth by exchanging messages for a fixed period of time rather than a fixed number of ex-

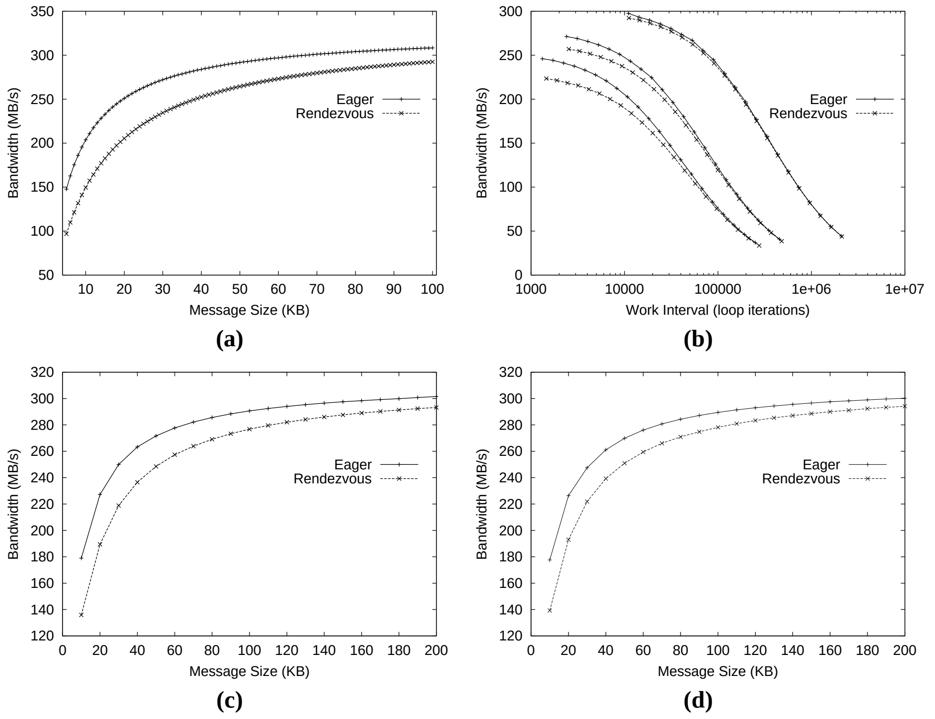

Figure 1. Bandwidth comparisons of rendezvous and eager for: **(a)** Ping-pong, **(b)** COMB PWW, **(c)** NetPIPE ping-pong, and **(d)** NetPIPE pipeline

changes. By default, NetPIPE does not insure that receives are pre-posted, but it does have an option to enable this feature. We used the default settings and did not enable pre-posted receives for our tests. In addition to the ping-pong strategy, NetPIPE also has a pipelined test that sends several messages in a burst. Figure 1(c) shows the performance of the ping-pong version, while Figure 1(d) shows the pipelined version. As with the previous benchmarks, the eager-rendezvous protocol signifi cantly outperforms the standard-rendezvous protocol for all message sizes.

5 New Micro-Benchmark

Benchmark results in Section 4 indicate that the eager-rendezvous protocol could provide signifi cantly more performance to applications. However, only the NetPIPE benchmark considers the possibility of unexpected messages. Real applications are likely to have a mixture of pre-posted and unexpected messages. It follows that the eager-rendezvous protocol would be benefi cial to applications that pre-post a majority of their receives, while the standard-rendezvous protocol would be more appropriate for applications that do not.

In order to test this hypothesis, we designed and implemented a micro-benchmark that measures bandwidth performance with a parameterized percentage of pre-posted

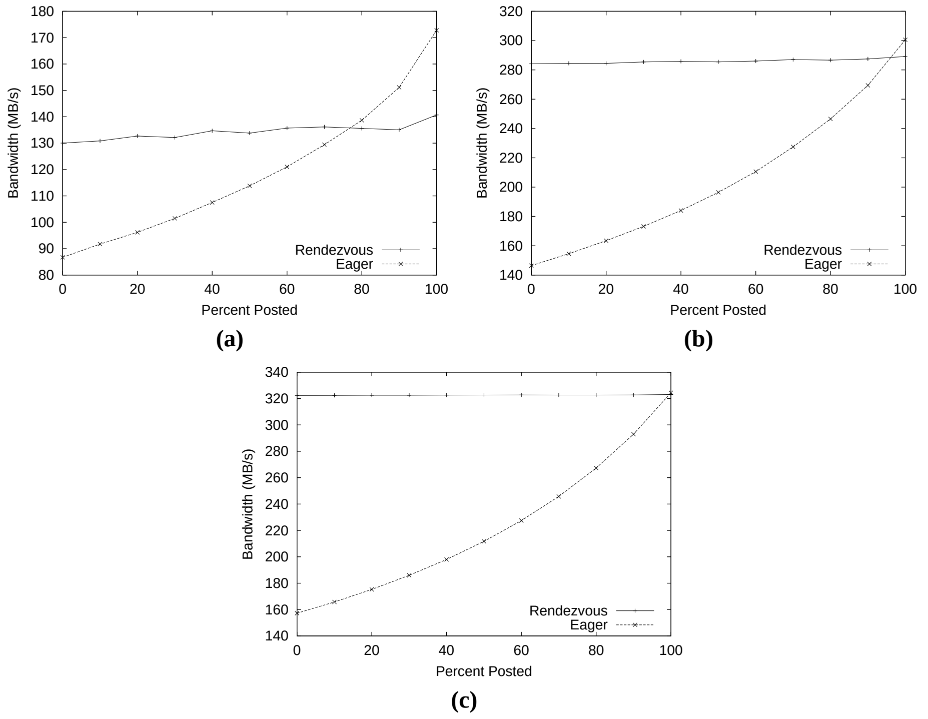

Figure 2. Unexpected message bandwidth for **(a)** 10 KB, **(b)** 100 KB, and **(c)** 1 MB messages

receives. This micro-benchmark allows us to analyze bandwidth performance for both protocols while varying the number of unexpected messages. The results of this new micro-benchmark are displayed in Figure 2. The results are somewhat surprising. For 10 KB messages shown in Figure 2(a), the standard-rendezvous protocol outperforms the eager-rendezvous protocol until nearly 80% of the receives are pre-posted. As the message size is increased, the standard-rendezvous protocol continues to signifi cantly outperform the eager-rendezvous version. The eager-rendezvous protocol is only more effective when more than 90% of 100 KB message are pre-posted, and when nearly all of the 1 MB messages are pre-posted.

6 Summary

In this paper, we have described an implementation of a standard-rendezvous protocol for long messages and compared its performance against a rendezvous protocol with an eager optimization. Our initial premise was that the eager optimization could provide a possible performance advantage for applications that pre-post receives.

We described the extra complexity of the standard-rendezvous protocol. It must maintain a list of posted receives and must continually try to process receives, even while waiting to complete send operations. This can potentially be less effi cient, since progress can only be made on outstanding requests when the application makes an MPI library call.

We compared the performance of these two different long message protocols using three different micro-benchmarks. We then designed and implemented a fourth micro-benchmark that attempts to provide more insight into how these protocols would perform in the context of real applications. Surprisingly, the results of this more realistic benchmark indicate that the eager optimization only outperforms the standard-rendezvous protocol if a large percentage of receive operations are pre-posted.

7 Future Work

We are continuing to evaluate the two different protocols presented in this paper for real applicaions. We have instrumented our MPI implementation to keep track of expected and unexpected messages, and we hope to be able to correlate the performance of the different protocols with the percentage of pre-posted receives. We are also trying to understand trends as applications are run on increasing numbers of nodes. We also hope to quantify other factors, such as the dependence on progress independent of making MPI calls, and their impact on the performance of these two protocols.

We believe that this work will help us to better understand the message passing characteristics of our important applications. We hope to use information about application behavior, MPI protocols, and resource usage in our design of the network hardware and software for the follow-on to the ASCI Red machine, called Red Storm. This machine is a joint project between Cray, Inc. and Sandia. The hardware and software architecture of this new machine is very similar to ASCI Red, so this data should be of great benefi t.

References

1. Ron Brightwell and Lance Shuler. Design and Implementation of MPI on Puma Portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.
2. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.
3. William Lawry, Christopher Wilson, Arthur B. Maccabe, and Ron Brightwell. Comb: A portable benchmark suite for assessing mpi overlap. Technical Report TR-CS-2002-13, Computer Science Department, The University of New Mexico, April 2002.
4. Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B. Maccabe, Lee Ann Fisk, and T. Mack Stallcup. The Puma Operating System for Massively Parallel Computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.
5. Q.O. Snell, A. Mikler, and J.L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator . In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.
6. Stephen R. Wheat Timothy G. Mattson, David Scott. A TeraFLOPS Supercomputer in 1996: The ASCI TFLOP System. In *Proceedings of the 1996 International Parallel Processing Symposium*, 1996.
7. F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proceedings of SC'99*, November 1999.